Probabilistic Recursion Theory and Implicit Computational Complexity

Ugo Dal Lago, Sara Zuppiroli

Probabilistic Recursion Theory and Implicit Computational Complexity*

Ugo Dal Lago and Sara Zuppiroli

Università di Bologna & INRIA
{dallago,zuppirol}@cs.unibo.it

Abstract. We show that probabilistic computable functions, i.e., those functions outputting distributions and computed by probabilistic Turing machines, can be characterized by a natural generalization of Church and Kleene's partial recursive functions. The obtained algebra, following Leivant, can be restricted so as to capture the notion of polytime sampleable distributions, a key concept in average-case complexity and cryptography.

1 Introduction

Models of computation as introduced one after the other in the first half of the last century were all designed around the assumption that *determinacy* is one of the key properties to be modeled: given an algorithm and an input to it, the sequence of computation steps leading to the final result is *uniquely* determined by the way an *algorithm* describes the state evolution. The great majority of the introduced models are *equivalent*, in that the classes of functions (on, say, natural numbers) they are able to compute are the same.

The second half of the 20th century has seen the assumption above relaxed in many different ways. Nondeterminism, as an example, has been investigated as a way to abstract the behavior of certain classes of algorithms, this way facilitating their study without necessarily changing their expressive power: think about how NFAs [15] make the task of proving closure properties of regular languages easier.

A relatively recent step in this direction consists in allowing algorithms' internal state to evolve probabilistically: the next state is not *functionally* determined by the current one, but is obtained from it by performing a process having possibly many outcomes, each with a certain probability. Again, probabilistically evolving computation can be a way to abstract over determinism, but also a way to model situations in which algorithms have access to a source of true randomness.

Probabilistic models are nowadays more and more pervasive. Not only are they a formidable tool when dealing with uncertainty and incomplete information, but they sometimes are a *necessity* rather than an option, like in computational cryptography (where, e.g., secure public key encryption schemes need

* This work is partially supported by the ANR project 12IS02001 PACE.

to be probabilistic [9]). A nice way to deal computationally with probabilistic models is to allow probabilistic choice as a primitive when designing algorithms, this way switching from usual, deterministic computation to a new paradigm, called probabilistic computation.

But what does the presence of probabilistic choice give us in terms of expressivity? Are we strictly more expressive than usual, deterministic, computation? And how about efficiency: is it that probabilistic choice permits to solve computational problems more efficiently? These questions have been among the most central in the theory of computation, and in particular in computational complexity, in the last forty years (see below for more details about related work). Roughly, while probability has been proved not to offer any advantage in the absence of resource constraints, it is not known whether probabilistic classes such as **BPP** or **ZPP** are different from **P**.

This work goes in a somehow different direction: we want to study probabilistic computation without necessarily *reducing* or *comparing* it to deterministic computation. The central assumption here is the following: a probabilistic algorithm computes what we call a *probabilistic function*, i.e. a function from a discrete set (e.g. natural numbers or binary strings) to *distributions* over the same set. What we want to do is to study the set of those probabilistic functions which can be computed by algorithms, possibly with resource constraints.

We give some initial results here. First of all, we provide a characterization of computable probabilistic functions by the natural generalization of Kleene's partial recursive functions, where among the initial functions there is now a function corresponding to tossing a fair coin. In the non-trivial proof of completeness for the obtained algebra, Kleene's minimization operator is used in an unusual way, making the usual proof strategy for Kleene's Normal Form Theorem (see, e.g., [18]) useless. We later hint at how to recover the latter by replacing minimization with a more powerful operator. We also mention how probabilistic recursion theory offers characterizations of concepts like the one of a computable distribution and of a computable real number.

The second part of this paper is devoted to applying the aforementioned recursion-theoretical framework to polynomial-time computation. We do that by following Bellantoni and Cook's and Leivant's works [1, 12], in which polynomial-time deterministic computation is characterized by a restricted form of recursion, called *predicative* or *ramified* recursion. Endowing Leivant's ramified recurrence with a random base function, in particular, is shown to provide a characterization of polynomial-time computable distributions, a key notion in average-case complexity [2].

Related Work. This work is rooted in the classic theory of computation, and in particular in the definition of partial computable functions as introduced by Church and later studied by Kleene [11]. Starting from the early fifties, various forms of automata in which probabilistic choice is available have been considered (e.g. [14]). The inception of probabilistic choice into an universal model of computation, namely Turing machines, is due to Santos [16, 17], but is (essentially) already there in an earlier work by De Leeuw and others [5]. Some years later,

Gill [6] considered probabilistic Turing machines with bounded complexity: his work has been the starting point of a florid research about the interplay between computational complexity and randomness. Among the many side effects of this research one can of course mention modern cryptography [10], in which algorithms (e.g. encryption schemes, authentication schemes, and adversaries for them) are almost invariably assumed to work in probabilistic polynomial time.

Implicit computational complexity (ICC), which studies machine-free characterizations of complexity classed based on mathematical logic and programming language theory, is a much younger research area. Its birth is traditionally made to correspond with the beginning of the nineties, when Bellantoni and Cook [1] and Leivant [12] independently proposed function algebras precisely characterizing (deterministic) polynomial time computable functions. In the last twenty years, the area has produced many interesting results, and complexity classes spanning from the logarithmic space computable functions to the elementary functions have been characterized by, e.g., function algebras, type systems [13], or fragments of linear logic [7]. Recently, some investigations on the interplay between implicit complexity and probabilistic computation have started to appear [3]. There is however an intrinsic difficulty in giving *implicit* characterizations of probabilistic classes like **BPP** or **ZPP**: the latter are semantic classes defined by imposing a polynomial bound on time, but also appropriate bounds on the probability of error. This makes the task of enumerating machines computing problems in the classes much harder and, ultimately, prevents from deriving implicit characterization of the classes above. Again, our emphasis is different: we do not see probabilistic algorithms as artifacts computing functions of the same kind as the one deterministic algorithms compute, but we see probabilistic algorithms as devices outputing distributions.

2 Probabilistic Recursion Theory

In this section we provide a characterization of the functions computed by a Probabilistic Turing Machine (PTM) in terms of a function algebra *à la* Kleene. We first define *probabilistic recursive functions*, which are the elements of our algebra. Next we define formally the class of probabilistic functions computed by a PTM. Finally, we show the equivalence of the two introduced classes. In the following, $\mathbb{R}_{[0,1]}$ is the unit interval.

Since PTMs compute probability distributions, the functions that we consider in our algebra have domain $\mathbb{N}^k$ and codomain $\mathbb{N} \to \mathbb{R}_{[0,1]}$ (rather than $\mathbb{N}$ as in the classic case). The idea is that if $f(x)$ is a function which returns $r \in \mathbb{R}_{[0,1]}$ on input $y \in \mathbb{N}$, then r is the probability of getting y as the output when feeding f with the input x. We note that we could extend our codomain from $\mathbb{N} \to \mathbb{R}_{[0,1]}$ to $\mathbb{N}^k \to \mathbb{R}_{[0,1]}$, however we use $\mathbb{N} \to \mathbb{R}_{[0,1]}$ in order to simplify the presentation.

Definition 1 (Pseudodistributions and Probabilistic Functions). *A* pseudodistribution *on* $\mathbb{N}$ *is a function* $\mathcal{D} : \mathbb{N} \to \mathbb{R}_{[0,1]}$ *such that* $\sum_{n\in\mathbb{N}} \mathcal{D}(n) \leq 1$. $\sum_{n\in\mathbb{N}} \mathcal{D}(n)$ *is often denoted as* $\sum \mathcal{D}$. *Let* $\mathbb{P}_{\mathbb{N}}$ *be the set of all pseudodistributions on* $\mathbb{N}$. *A* probabilistic function *(PF) is a function from* $\mathbb{N}^k$ *to* $\mathbb{P}_{\mathbb{N}}$, *where*

$\mathbb{N}^k$ *stands for the set of* k*-tuples in* $\mathbb{N}$*. We use the expression* $\{n_1^{p1}, \ldots, n_k^{pk}\}$ *to denote the pseudodistribution* $\mathcal{D}$ *defined as* $\mathcal{D}(n) = \sum_{n_i=n} p_i$*. Observe that* $\sum \mathcal{D} = \sum_{i=1}^{k} p_i$*. When this does not cause ambiguity, the terms distribution and pseudodistribution will be used interchangeably.*

Please notice that probabilistic functions are always *total* functions, but their codomain is a set of distributions which do not necessarily sum to 1, but rather to a real number *smaller* or equal to 1, this way modeling the probability of divergence. For example, the nowhere-defined partial function $\Omega : \mathbb{N} \rightharpoonup \mathbb{N}$ of classic recursion theory becomes a probabilistic function which returns the empty distributions $\emptyset$ on any input. The first step towards defining our function algebra consists in giving a set of functions to start from:

Definition 2 (Basic Probabilistic Functions). *The* basic probabilistic functions *(BPFs) are as follows:*

- *The* zero function $z : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *defined as:* $z(n)(0) = 1$ *for every* $n \in \mathbb{N}$*;*
- *The* successor function $s : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *defined as:* $s(n)(n+1) = 1$ *for every* $n \in \mathbb{N}$*;*
- *The* projection function $\Pi_m^n : \mathbb{N}^n \to \mathbb{P}_{\mathbb{N}}$ *defined as:* $\Pi_m^n(k_1, \cdots, k_n)(k_m) = 1$ *for every positive* $n, m \in \mathbb{N}$ *such that* $1 \leq m \leq n$*;*
- *The* fair coin function $r : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *that is defined as:*

$$r(x)(y) = \begin{cases} 1/2 \text{ if } y = x \\ 1/2 \text{ if } y = x+1 \end{cases}$$

The first three BPFs are the same as the basic functions from classic recursion theory, while *rand* is the only truly probabilistic BPF.

The next step consists in defining how PFs *compose*. Function composition of course cannot be used here, because when composing two PFs g and f the codomain of g does not match with the domain of f. Indeed g returns a distribution $\mathbb{N} \to \mathbb{R}_{[0,1]}$ while f expects a natural number as input. What we have to do here is the following. Given an input $x \in \mathbb{N}$ and an output $y \in \mathbb{N}$ for the composition $f \bullet g$, we apply the distribution $g(x)$ to any value $z \in \mathbb{N}$. This gives a probability $g(x)(z)$ which is then multiplied by the probability that the distribution $f(z)$ associates to the value $y \in \mathbb{N}$. If we then consider the sum of the obtained product $g(x)(z) \cdot f(z)(y)$ on all possible $z \in \mathbb{N}$ we obtain the probability of $f \bullet g$ returning y when fed with x. The sum is due to the fact that two different values, say $z_1, z_2 \in \mathbb{N}$, which provide two different distributions $f(z_1)$ and $f(z_2)$ must both contribute to the same probability value $f(z_1)(y) + f(z_2)(y)$ for a specific y. In other words, we are doing nothing more than lifting f to a function from distributions to distributions, then composing it with g. Formally:

Definition 3 (Composition). *We define the* composition $f \bullet g : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *of two functions* $f : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *and* $g : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *as:*

$$((f \bullet g)(x))(y) = \sum_{z \in \mathbb{N}} g(x)(z) \cdot f(z)(y).$$

The previous definition can be generalized to functions taking more than one parameter in the expected way:

Definition 4 (Generalized Composition). *We define the* generalized composition *of functions* $f : \mathbb{N}^n \to \mathbb{P}_\mathbb{N}$, $g_1 : \mathbb{N}^k \to \mathbb{P}_\mathbb{N}, \ldots, g_n : \mathbb{N}^k \to \mathbb{P}_\mathbb{N}$ *as the function* $f \odot (g_1, \ldots, g_n) : \mathbb{N}^k \to \mathbb{P}_\mathbb{N}$ *defined as follows:*

$$((f \odot (g_1, \ldots, g_n))(\mathbf{x}))(y) = \sum_{z_1, \ldots, z_n \in \mathbb{N}} \left(f(z_1, \ldots, z_n)(y) \cdot \prod_{1 \leq i \leq n} g_i(\mathbf{x})(z_i) \right).$$

With a slight abuse of notation, we can treat probabilistic functions as ordinary functions when forming expressions. Suppose, as an example, that $x \in \mathbb{N}$ and that $f : \mathbb{N}^3 \to \mathbb{P}_\mathbb{N}$, $g : \mathbb{N} \to \mathbb{P}_\mathbb{N}$, $h : \mathbb{N} \to \mathbb{P}_\mathbb{N}$. Then the expression $f(g(x), x, h(x))$ stands for the distribution in $\mathbb{P}_\mathbb{N}$ defined as follows: $(f \odot (g, id, h))(x)$, where $id = \Pi^1_1$ is the identity PF.

The way we have defined probabilistic functions and their composition is reminiscent of, and indeed inspired by, the way one defines the Kleisli category for the Giry monad, starting from the category of partial functions on sets. This categorical way of seeing the problem can help a lot in finding the right definition, but by itself is not adequate to proving the existence of a correspondence with machines like the one we want to give here.

Primitive recursion is defined as in Kleene's algebra, provided one uses composition as previously defined:

Definition 5 (Primitive Recursion). *Given functions* $g : \mathbb{N}^{k+2} \to \mathbb{P}_\mathbb{N}$, *and* $f : \mathbb{N}^k \to \mathbb{P}_\mathbb{N}$, *the function* $h : \mathbb{N}^{k+1} \to \mathbb{P}_\mathbb{N}$ *defined as*

$$h(\mathbf{x}, 0) = f(\mathbf{x}); \qquad h(\mathbf{x}, y+1) = g(\mathbf{x}, y, h(\mathbf{x}, y));$$

is said to be defined by primitive recursion *from* f *and* g, *and is denoted as* $rec(f, g)$.

We now turn our attention to the minimization operator which, as in the deterministic case, is needed in order to obtain the full expressive power of (P)TMs. The definition of this operator is in our case delicate and requires some explanation. Recall that, in the classic case, the minimization operator allows from a partial function $f : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$, to define another partial function, call it μf, which computes from $\mathbf{x} \in \mathbb{N}^k$ the least value of y such that $f(\mathbf{x}, y)$ is equal to 0, if such a value exists (and is undefined otherwise). In our case, again, we are concerned with distributions, hence we cannot simply consider the least value on which f returns 0, since functions return 0 *with a certain probability*. The idea is then to define the minimization μf as a function which, given an input $\mathbf{x} \in \mathbb{N}^k$, returns a distribution associating to each natural y the probability that the result of $f(\mathbf{x}, y)$ is 0 *and* the result of $f(\mathbf{x}, z)$ is positive for every $z < y$. Formally:

Definition 6 (Minimization). *Given a PF $f : \mathbb{N}^{k+1} \to \mathbb{P}_{\mathbb{N}}$, we define another PF $\mu f : \mathbb{N}^k \to \mathbb{P}_{\mathbb{N}}$ as follows:*

$$\mu f(\mathbf{x})(y) = f(\mathbf{x}, y)(0) \cdot (\prod_{z<y} (\sum_{k>0} f(\mathbf{x}, z)(k))).$$

We are finally able to define the class of functions we are interested in as follows.

Definition 7 (Probabilistic Recursive Functions). *The class $\mathscr{PR}$ of* probabilistic recursive functions *is the smallest class of probabilistic functions that contains the BPFs (Definition 2) and is closed under the operation of General Composition (Definition 4), Primitive Recursion (Definition 5) and Minimization (Definition 6).*

It is easy to show that $\mathscr{PR}$ includes all partial recursive functions, seen as probabilistic functions: first, for every partial function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$, define $p_f : \mathbb{N}^k \to \mathbb{P}_{\mathbb{N}}$ by stipulating that $p_f(\mathbf{x})(y) = 1$ whenever $y = f(\mathbf{x})$, and $p_f(\mathbf{x})(y) = 0$ otherwise; then, by an easy induction, $p_f \in \mathscr{PR}$ whenever f is partial recursive.

Example 1. The following are examples of probabilistic recursive functions:

- The *identity function* $id : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$, defined as $id(x)(x) = 1$. For all $x, y \in \mathbb{N}$ we have that
$$id(x)(y) = \begin{cases} 1 \text{ if } y = x \\ 0 \text{ otherwise} \end{cases}$$
as a consequence $id = \Pi^1_1$, and, since the latter is a BPF (Definition 2) id is in $\mathscr{PR}$.
- The probabilistic function $rand : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ such that for every $x \in \mathbb{N}$, $rand(x)(0) = \frac{1}{2}$ and $rand(x)(1) = \frac{1}{2}$ can be easily shown to be recursive, since $rand = r \odot z$ (and we know that both r and z are BPF). Actually, *rand* could itself be taken as the only genuinely probabilistic BPF, i.e., r can be constructed from *rand* and the other BPF by composition and primitive recursion. We proceed by defining $g : \mathbb{N}^3 \to \mathbb{P}_{\mathbb{N}}$ as follow:
$$g(x_1, x_2, z)(y) = \begin{cases} 1 \text{ if } y = z + 1 \\ 0 \text{ otherwise} \end{cases}$$
g is in $\mathscr{PR}$ because $g = s \odot (\Pi^3_3)$. Now we observe that the function *add* defined by $add(x, 0) = id(x)$ and $add(x_1, x_2 + 1) = g(x_1, x_2, add(x_1, x_2))$ is a probabilistic recursive function, since it can be obtained from basic functions using composition and primitive recursion. We can conclude by just observing that $r = add \odot (\Pi^1_1, rand)$.
- All functions we have proved recursive so far have the property that the returned distribution is *finite* for any input. Indeed, this is true for every probabilistic *primitive* recursive function, since minimization is the only way to break this form of finiteness. Consider the function $f : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ defined as $f(x)(y) = \frac{1}{2^{y-x+1}}$ if $y \geq x$, and $f(x)(y) = 0$ otherwise. We define another function $h : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ by stipulating that $h(x)(y) = \frac{1}{2^{y+1}}$ for every $x, y \in \mathbb{N}$. h

is a probabilistic recursive function; indeed consider the function $k : \mathbb{N}^2 \to \mathbb{P}_{\mathbb{N}}$ defined as $rand \odot \Pi_1^2$ and build $\mu\ k$. By definition,

$$(\mu\ k)(x)(y) = k(x,y)(0) \cdot (\prod_{z<y} (\sum_{q>0} k(x,z)(q))). \tag{1}$$

Then observe that $(\mu\ k)(x)(y) = \frac{1}{2^{y+1}}$: by (1), $(\mu\ k)(x)(y)$ unfolds into a product of exactly $y+1$ copies of $\frac{1}{2}$, each "coming from the flip of a distinct coin". Hence, $h = \mu\ k$. Then we observe that

$$(add \odot (\mu\ k, id))(x)(y) = \sum_{z_1,z_2} add(z_1,z_2)(y) \cdot ((\mu\ k)(x)(z_1) \cdot id(x)(z_2)).$$

But notice that $id(x)(z_2) = 1$ only when $z_2 = x$ (and in the other cases $id(x)(z_2) = 0$), $(\mu\ k)(x)(z_1) = \frac{1}{2^{z_1+1}}$, and $add(z_1,z_2)(y) = 1$ only when $z_1 + z_2 = y$ (and in the other cases, $add(z_1,z_2)(y) = 0$). This implies that the term in the sum is different from 0 only when $z_2 = x$ and $z_1 + z_2 = y$, namely when $z_1 = y - z_2 = y - x$, and in that case its value is $\frac{1}{2^{y-x+1}}$. Thus, we can claim that $f = (add \odot (\mu\ k, id))$, and that f is in $\mathscr{PR}$.

2.1 Probabilistic Turing Machines and Computable Functions

In this section we introduce computable functions as those probabilistic functions which can be computed by Probabilistic Turing Machines. As previously mentioned, probabilistic computation devices have received a wide interest in computer science already in the fifties [5] and early sixties [14]. A natural question which arose was then to see what happened if random elements were allowed in a Turing machine. This question led to several formalizations of probabilistic Turing machines (PTMs in the following) [5, 16] — which, essentially, are Turing machines which have the ability to flip coins in order to make random decisions — and to several results concerning the computational complexity of problems when solved by PTMs [6].

Following [6], a Probabilistic Turing Machine (PTM) M can be seen as a Turing Machine with two transition functions δ_0, δ_1. At each computation step, either δ_0 or δ_1 can be applied, each with probability $1/2$. Then, in a way analogous to the deterministic case, we can define a notion of a (initial, final) configuration for a PTM M. In the following, Σ_b denotes the set of possible symbols on the tape, including a blank symbol $\Box$; Q denotes the set of states; $Q_f \subseteq Q$ denotes the set of final states and $q_s \in Q$ denotes the initial state.

Definition 8 (Probabilistic Turing Machine). *A Probabilistic Turing Machine (PTM) is a Turing machine endowed with two transition functions δ_0, δ_1. At each computation step the transition function δ_0 can be applied with probability $1/2$ and the transition δ_1 can be applied with probability $1/2$.*

Definition 9 (Configuration of a PTM). *Let M be a PTM. We define a PTM* configuration *as a 4-tuple $\langle s, a, t, q\rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$ such that:*

- *The first component, $s \in \Sigma_b^*$, is the portion of the tape lying on the left of the head.*
- *The second component, $a \in \Sigma_b$, is the symbol the head is reading.*
- *The third component, $t \in \Sigma_b^*$, is the portion of the tape lying on the right of the head.*
- *The fourth component, $q \in Q$ is the current state.*

Moreover we define the set of all configurations as $\mathcal{C}_M = \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$.

Definition 10 (Initial and Final Configurations of a PTM). *Let M be a PTM. We define the* initial configuration *of M for the string s as the configuration in the form $\langle \varepsilon, a, v, q_s \rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$ such that $s = a \cdot v$ and the fourth component, $q_s \in Q$, is the initial state. We denote it with $\mathcal{IN}_M^s$. Similarly, we define a* final configuration *of M for s as a configuration $\langle s, \Box, \varepsilon, q_f \rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q_f$. The set of all such final configurations for a PTM M is denoted by $\mathcal{FC}_M^s$.*

For a function $T : \mathbb{N} \to \mathbb{N}$, we say that a PTM M *runs in time bounded by* T if for any input x, M halts on input x within $T(|x|)$ steps *independently* of the random choices it makes. Thus, M *works in polynomial time* if it runs in time bounded by P, where P is any polynomial.

Intuitively, the function computed by a PTM M associates to each input s, a pseudodistribution which indicates the probability of reaching a final configuration of M from $\mathcal{IN}_M^s$. It is worth noticing that, differently from the deterministic case, since in a PTM the same configuration can be obtained by different computations, the probability of reaching a given final configuration is the *sum* of the probabilities of reaching the configuration along all computation paths, of which there can be (even infinitely) many. It is thus convenient to define the function computed by a PTM through a fixpoint construction, as follows. First, we can define a partial order on the string distributions as follows.

Definition 11. *A* string pseudodistribution *on Σ^* is a function $\mathcal{D} : \Sigma^* \to \mathbb{R}_{[0,1]}$ such that $\sum_{s \in \Sigma^*} \mathcal{D}(s) \leq 1$. $\mathbb{P}_{\Sigma^*}$ denotes the set of all string pseudodistributions on Σ^*. The relation $\sqsubseteq_{\mathbb{P}_{\Sigma^*}} \subseteq \mathbb{P}_{\Sigma^*} \times \mathbb{P}_{\Sigma^*}$ is defined as the pointwise extension of the usual partial order on $\mathbb{R}$.*

It is easy to show that the relation $\sqsubseteq_{\mathbb{P}_{\Sigma^*}}$ from Definition 11 is a partial order. Next, we can define the domain $\mathcal{CEV}$ of those functions computed by a PTM M from a given configuration, i.e., the set of those functions f such that $f : \mathcal{C}_M \to \mathbb{P}_{\Sigma^*}$. Inheriting the structure from $\mathbb{P}_{\Sigma^*}$, we can obtain a poset $(\mathcal{CEV}, \sqsubseteq_{\mathcal{CEV}})$, again by defining $\sqsubseteq_{\mathcal{CEV}}$ pointwise. Moreover, it is also easy to show that the two introduced posets are ω**CPO**s.

We can now define a functional F_M on $\mathcal{CEV}$ which will be used to define the function computed by M via a fixpoint construction. Intuitively, the application of the functional F_M describes *one* computation step. Formally:

Definition 12. *Given a PTM M, we define a functional $F_M : \mathcal{CEV} \to \mathcal{CEV}$ as:*

$$F_M(f)(C) = \begin{cases} \{s^1\} & \text{if } C \in \mathcal{FC}_M^s; \\ \frac{1}{2} f(\delta_0(C)) + \frac{1}{2} f(\delta_1(C)) & \text{otherwise.} \end{cases}$$

One can show that the functional F_M from Definition 12 is continuous on $\mathcal{CEV}$. A classic fixpoint theorem ensures that F_M has a least fixpoint. Such a least fixpoint is, once composed with a function returning $\mathcal{IN}_M^s$ from s, the *function computed by the machine* M, which is denoted as $\mathcal{IO}_M : \Sigma^* \to \mathbb{P}_{\Sigma^*}$. The set of those functions which can be computed by any PTM is denoted as $\mathscr{PC}$, while $\mathscr{PPC}$ is the set of probabilistic functions which can be computed by a PTM working in *polynomial* time. The notion of a computable probabilistic function subsumes other key notions in probabilistic and real-number computation. As an example, *computable distributions* can be characterized as those distributions on Σ^* which can be obtained as the result of a function in $\mathscr{PC}$ on a *fixed* input. Analogously, *computable real numbers* from the unit interval $[0, 1]$ can be seen as those elements of $\mathbb{R}$ in the form $f(0)(0)$ for a computable function $f \in \mathscr{PC}$.

2.2 Probabilistic Recursive Functions equals Functions computed by Probabilistic Turing Machines

In this section we prove that probabilistic *recursive* functions are the same as probabilistic *computable* functions, modulo an appropriate bijection between strings and natural numbers which we denote (as its inverse) with $\overline{(\cdot)}$.

In order to prove the equivalence result we first need to show that a probabilistic recursive function can be computed by a PTM. This result is not difficult and, analogously to the deterministic case, is proved by exhibiting PTMs which simulate the basic probabilistic recursive functions and by showing that $\mathscr{PC}$ is closed by composition, primitive recursion, and minimization. We omit the details, which can be found in [4].

The most difficult part of the equivalence proof consists in proving that each probabilistic computable function is actually *recursive*. Analogously to the classic case, a good strategy consists in representing configurations as natural numbers, then encoding the transition functions of the machine at hand, call it M, as a (recursive) function on $\mathbb{N}$. In the classic case the proof proceeds by making essential use of the minimization operator by which one determines the *number* of transition steps of M necessary to reach a final configuration, if such number exists. This number can then be fed into another function which simulates M (on an input) a given number of steps, and which is primitive recursive. In our case, this strategy does not work: the number of computation steps can be infinite, even when the convergence probability is 1. Given our definition of minimization which involves distributions, this is delicate, since we have to define a suitable function on the PTM computation tree to be minimized.

In order to adapt the classic proof, we need to formalize the notion of a *computation tree* which represents all computation paths corresponding to a given input string x. We define such a tree as follows. Each node is labelled by a configuration of the machine and each edge represents a computation step. The root is labelled with $\mathcal{IN}_M^x$ and each node labelled with C has either no child (if C is final) or 2 children (otherwise), labelled with $\delta_0(C)$ and $\delta_1(C)$. Please notice that the same configuration may be duplicated across a single level of the

tree as well as appear at different levels of the tree; nevertheless we represent each such appearance by a separate node.

We can naturally associate a probability with each node, corresponding to the probability that the node is reached in the computation: it is $\frac{1}{2^n}$, where n is the height of the node. The probability of a particular *final* configuration is the sum of the probabilities of all leaves labelled with that configuration. We also enumerate nodes in the tree, top-down and from left to right, by using binary strings in the following way: the root has associated the number ε. Then if b is the binary string representing the node N, the left child of N has associated the string $b \cdot 0$ while the right child has the number $b \cdot 1$. Note that from this definition it follows that each binary number associated to a node N indicates a path in the tree from the root to N. The computation tree for x will be denoted as $CT_M(x)$

We give now a more explicit description of the constructions described above. First we need to encode the rational numbers $\mathbb{Q}$ into $\mathbb{N}$. Let $pair : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be any recursive bijection between pairs of natural numbers and natural numbers such that *pair* and its inverse are both computable. Let then *enc* be just p_{pair}, i.e. the function $enc : \mathbb{N} \times \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ defined as follows

$$enc(a, b)(q) = \begin{cases} 1 & \text{if } q = pair(a, b) \\ 0 & \text{otherwise} \end{cases}$$

The function *enc* allows to represent positive rational numbers as pairs of natural numbers in the obvious way and is recursive.

It is now time to define a few notions on computation trees

Definition 13 (Computation Trees and String Probabilities). *The function $PT_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ is defined by stipulating that $PT_M(x, y)$ is the probability of observing the string $\overline{y}$ in the tree $CT_M(x)$, namely $\frac{1}{2^{|\overline{y}|}}$.*

Of course, PT_M is partial recursive, thus p_{PT_M} is probabilistic recursive. Since the same configuration C can label more than one node in a computation tree $CT_M(x)$, PT_M does not indicate the probability of reaching C, even when C is the label of the node corresponding to the second argument. Such a probability can be obtained by summing the probability of all nodes labelled with the configuration at hand:

Definition 14 (Configuration Probability). *Suppose given a PTM M. If $x \in \mathbb{N}$ and $z \in \mathcal{C}_M$, the subset $CC_M(x, z)$ of $\mathbb{N}$ contains precisely the indices of nodes of $CT_M(x)$ which are labelled by z. The function $PC_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ is defined as follows:*

$$PC_M(x, z) = \Sigma_{y \in CC_M(x,z)} PT_M(x, y)$$

Contrary to PT_M, there is nothing guaranteeing that PC_M is indeed computable. In the following, however, what we do is precisely to show that this is the case.

In Figure 1 we show an example of computation tree $CT_M(x)$ for an hypothetical PTM M and an input x. The leaves, depicted as red nodes, represent

the final configurations of the computation. So, for example, $PC_M(x, C) = 1$, while $PC_M(x, E) = \frac{3}{4}$. Indeed, notice that there are three nodes in the tree which are labelled with E, namely those corresponding to the binary strings 00, 01, and 10. As we already mentioned, our proof separates the classic part of

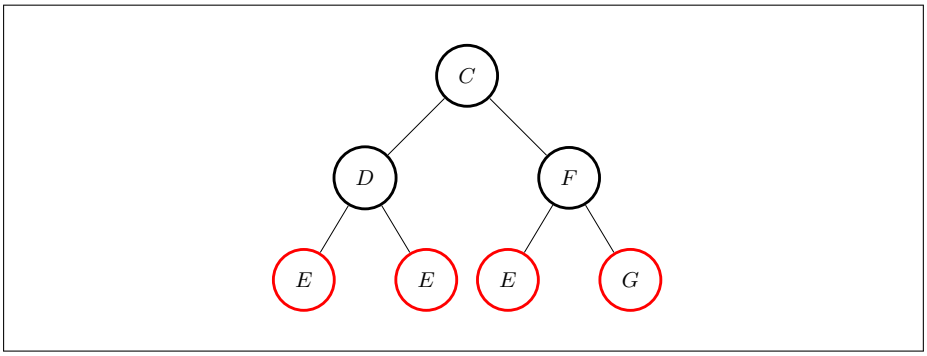

Fig. 1. An Example of a Computation Tree

the computation performed by the underlying PTM, which essentially computes the configurations reached by the machine in different paths, from the probabilistic part, which instead computes the probability values associated to each computation by using minimization. These two tasks are realized by two suitable probabilistic recursive functions, which are then composed to obtain the function computed by the underlying PTM. We start with the probabilistic part, which is more complicated.

We need to define a function, which returns the *conditional* probability of terminating at the node corresponding to the string $\overline{y}$ in the tree $CT_M(x)$, given that all the nodes $\overline{z}$ where $z < y$ are labelled with non-final configurations. This is captured by the following definition:

Definition 15. *Given a PTM M, we define $PT^0_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ and $PT^1_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ as follows:*

$$PT^1_M(x, y) = \begin{cases} 1 & \text{if } y \text{ is not a leaf of } CT_M(x); \\ 1 - PT^0_M(x, y) & \text{otherwise;} \end{cases}$$

$$PT^0_M(x, y) = \begin{cases} 0 & \text{if } y \text{ is not a leaf of } CT_M(x); \\ \frac{PT_M(x,y)}{\prod_{k<y} PT^1_M(x,k)} & \text{otherwise;} \end{cases}$$

Note that, according to previous definition, $PT^1_M(x, y)$ is the probability of *not* terminating the computation in the node y, while $PT^0_M(x, y)$ represents the probability of terminating the computation in the node y, both *knowing* that the computation has not terminated in any node k preceding y.

Proposition 1. *The functions* $PT^0_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ *and* $PT^1_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ *are partial recursive.*

Proof. Please observe that PT_M is partial recursive and that the definitions above are mutually recursive, but the underlying order is well-founded. Both functions are thus intuitively computable, thus partial recursive by the Church-Turing thesis. □

The reason why the two functions above are useful is because they associate the distribution $\{0^{PT^1_M(x,y)}, 1^{PT^0_M(x,y)}\}$ to each pair of natural numbers (x, y). In Figure 2, we give the quantities we have just defined for the tree from Figure 1. Each internal node is associated with the same distribution $\{0^0, 1^1\}$. Only the leaves are associated with nontrivial distributions. As an example, the distribution associated to the node 10 is $\{0^{1/2}, 1^{1/2}\}$, because we have that

$$\begin{aligned} PT^0_M(x, \overline{10}) &= \frac{PT_M(x, \overline{10})}{\prod_{k<\overline{10}} PT^1_M(x, k)} \\ &= \frac{1}{4 \cdot PT^1_M(x, \overline{01}) \cdot PT^1_M(x, \overline{00}) \cdot PT^1_M(x, \overline{1}) \cdot PT^1_M(x, \overline{0}) \cdot PT^1_M(x, \overline{\varepsilon})} \\ &= \frac{1}{4 \cdot PT^1_M(x, \overline{01}) \cdot PT^1_M(x, \overline{00})}. \end{aligned}$$

As it can be easily verified, $PT^1_M(x, \overline{00}) = \frac{3}{4}$, while $PT^1_M(x, \overline{01}) = \frac{2}{3}$. Thus, $PT^0_M(x, \overline{10}) = \frac{1}{2}$.

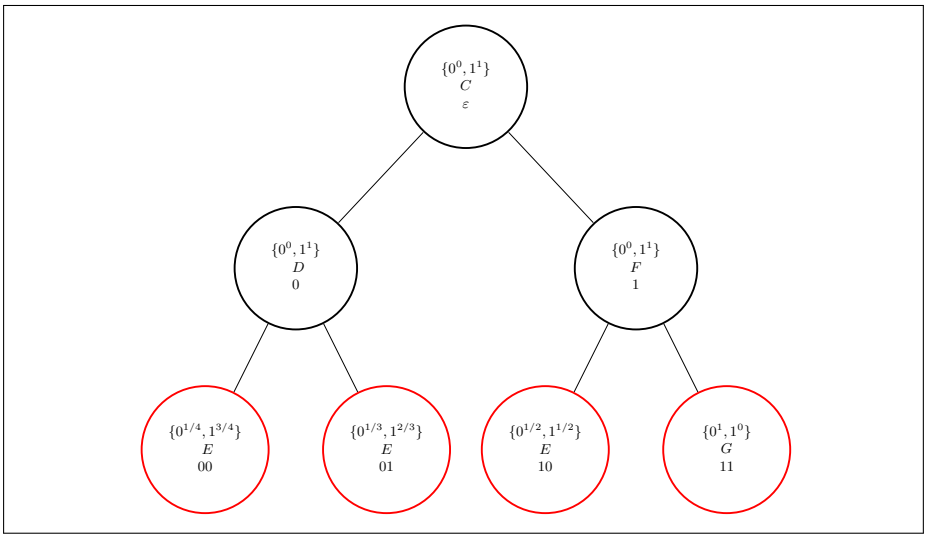

Fig. 2. The Conditional Probabilities for the Computation Tree from Figure 1

We now need to go further, and prove that the probabilistic function returning, on input (x, y), the distribution $\{0^{PT^1_M(x,y)}, 1^{PT^0_M(x,y)}\}$ is recursive. This is captured by the following definition:

Definition 16. *Given a PTM M, the function $PTC_M : \mathbb{N} \times \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ is defined as follows*

$$PTC_M(x,y)(z) = \begin{cases} PT^0_M(x,y) & \text{if } z = 0; \\ PT^1_M(x,y) & \text{if } z = 1; \\ 0 & \text{otherwise-} \end{cases}$$

The function PTC_M is really the core of our encoding. On the one hand, we will show that it is indeed recursive. On the other, minimizing it is going to provide us exactly with the function we need to reach our final goal, namely proving that the probabilistic function computed by M is itself recursive. But how should we proceed if we want to prove PTC_M to be recursive? The idea is to compose $p_{PT^1_M}$ with a function that turns its input into the probability of returning 1. This is precisely what the following function does:

Definition 17. *The function $I2P : \mathbb{Q} \to \mathbb{P}_{\mathbb{N}}$ is defined as follows*

$$I2P(x)(y) = \begin{cases} x & \text{if } (0 \leq x \leq 1) \wedge (y = 1) \\ 1 - x & \text{if } (0 \leq x \leq 1) \wedge (y = 0) \\ 0 & \text{otherwise} \end{cases}$$

Please observe how the input to *I2P* is the set of rational numbers, as usual encoded by pairs of natural numbers. Previous definitions allow us to treat (rational numbers representing) probabilities in our algebra of functions. Indeed:

Proposition 2. *The probabilistic function I2P is recursive.*

Proof. We first observe that $h : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ defined as

$$h(x)(y) = 1/2^{y+1}$$

is a probabilistic recursive function, because $h = \mu\,(rand \odot \Pi^2_1)$. Next we observe that every $q \in \mathbb{Q} \cap [0, 1]$ can be represented in binary notation as:

$$q = \sum_{i \in \mathbb{N}} \frac{c^q_i}{1/2^{i+1}}$$

where $c^q_i \in \{0, 1\}$ (i.e., c^q_i is the i-th element of the binary representation of q). Moreover, a function computing such a c^q_i from q and i is partial recursive. Hence we can define $b : \mathbb{N} \times \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ as follows

$$b(q,i)(y) = \begin{cases} 1 \text{ if } y = c^q_i \\ 0 \text{ otherwise} \end{cases}$$

and conclude that b is indeed a probabilistic recursive function (because $\mathscr{PR}$ includes all the partial recursive functions, seen as probabilistic functions). Observe that:

$$b(q,i)(y) = \begin{cases} c^q_i & \text{if } y = 1 \\ 1 - c^q_i & \text{if } y = 0 \end{cases}$$

From the definition of composition, it follows that

$$\begin{aligned}(b \odot (id, h))(q)(y) &= \sum_{x_1, x_2} b(x_1, x_2)(y) \cdot id(q)(x_1) \cdot h(q)(x_2) \\ &= \sum_{x_2} b(q, x_2)(y) \cdot h(q)(x_2) = \sum_{x_2} b(q, x_2)(y) \cdot \frac{1}{2^{x_2+1}} \\ &= \begin{cases} \sum_{x_2} \frac{c^q_{x_2}}{2^{x_2+1}} \text{ if } y = 1 \\ \sum_{x_2} \frac{1-c^q_{x_2}}{2^{x_2+1}} \text{ if } y = 0 \end{cases} = \begin{cases} q & \text{if } y = 1 \\ 1 - q & \text{if } y = 0 \end{cases}.\end{aligned}$$

This shows that

$$I2P = b \odot (id, h),$$

and hence that $I2P$ is probabilistic recursive. □

The following is an easy corollary of what we have obtained so far:

Proposition 3. *The probabilistic function PTC_M is recursive.*

Proof. Just observe that $PTC_M = I2P \odot p_{PT^1_M}$. □

The probabilistic recursive function obtained as the minimization of PTC_M allows to compute a probabilistic function that, given x, returns y with probability $PT_M(x, y)$ *if* y is a leaf (and otherwise the probability is just 0).

Definition 18. *The function $\mathcal{CF}_M : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ is defined as follows*

$$\mathcal{CF}_M(x)(y) = \begin{cases} PT_M(x, y) & \text{if } y \text{ corresponds to a leaf} \\ 0 & \text{otherwise.} \end{cases}$$

Proposition 4. *The probabilistic function $\mathcal{CF}_M$ is recursive.*

Proof. The probabilistic function $\mathcal{CF}_M$ is just the function obtained by minimizing PTC_M, which we already know to be recursive. Indeed, if z corresponds to a leaf, then:

$$\begin{aligned}(\mu PTC_M)(x)(z) &= PTC_M(x, z)(0) \cdot \prod_{y<z} \sum_{k>0} PTC_M(x, y)(k) \\ &= PTC_M(x, z)(0) \cdot \prod_{y<z} PTC_M(x, y)(1) \\ &= PT^0_M(x, z) \cdot \prod_{y<z} PT^1_M(x, y) \\ &= \frac{PT_M(x, z)}{\prod_{y<z} PT^1_M(x, y)} \cdot \prod_{y<z} PT^1_M(x, y) = PT_M(x, z).\end{aligned}$$

If, however, z does not correspond to a leaf, then:

$$\begin{aligned}(\mu PTC_M)(x)(z) &= PTC_M(x, z)(0) \cdot \prod_{y<z} \sum_{k>0} PTC_M(x, y)(k) \\ &= PT^0_M(x, z)(0) \cdot \prod_{y<z} \sum_{k>0} PTC_M(x, y)(k) = 0.\end{aligned}$$

This concludes the proof. □

We are almost ready to wrap up our result, but before proceeding further, we need to define the function $SP_M : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ that, given in input a pair (x, y) returns the (encoding) of the string found in the configuration labeling the node y in $CT_M(x)$. We can now prove the desired result:

Theorem 1. $\mathscr{PC} \subseteq \mathscr{PR}$.

Proof. It suffices to note that, given any PTM M, the function computed by M is nothing more than

$$p_{SP_M} \odot (id, \mathcal{CF}_M).$$

Indeed, one can easily realize that a way to simulate M consists in generating, from x, all strings corresponding to the leaves of $CT_M(x)$, each with an appropriate probability. This is indeed what $\mathcal{CF}_M$ does. What remains to be done is simulating p_{SP_M} along paths leading to final configurations. □

We are finally ready to prove the main result of this Section:

Corollary 1 $\mathscr{PR} = \mathscr{PC}$

Proof. Immediate from Theorem 1, observing that $\mathscr{PR} \subseteq \mathscr{PC}$ (this implication is easy to prove). □

The way we prove Corollary 1 implies that we cannot deduce Kleene's Normal Form Theorem from it: minimization has been used many times, some of them "deep inside" the construction. A way to recover Kleene's Theorem consists in replacing minimization with a more powerful operator, essentially corresponding to computing the fixpoint of a given function (see [4] for more details).

3 Characterizing Probabilistic Complexity by Tiering

In this section we provide a characterization of the probabilistic functions which can be computed in polynomial time by an algebra of functions acting on word algebras. More precisely, we define a type system inspired by Leivant's notion of tiering [12], which permits to rule out functions having a too-high complexity, thus allowing to isolate the class of *predicative probabilistic functions*. Finally, we give a hint at how the equivalence between polytime probabilistic functions and predicative probabilistic functions can be proved (more details are in [4]).

The constructions from Section 2 can be easily generalized to a function algebra on strings in a given alphabet Σ, which themselves can be seen as a *word algebra* $\mathbb{W}$. Base functions include a function computing the empty string, called ε, and concatenation with any character $a \in \Sigma$, called c_a. Projections remain of course available, while the only truly random functions concatenate a symbol $a \in \Sigma$ to the input, with probability $\frac{1}{2}$, or leave it unchanged, with probability $\frac{1}{2}$. Such a function is denoted as r_a. Composition and primitive recursion are available, although the latter takes the form of recursion *on notation*. We do not need minimization: the distribution a polytime computable probabilistic function

returns (on any input) is always finite, and primitive recursion is anyway powerful enough for our purposes.

The following construction is redundant in presence of primitive recursion, but becomes essential when predicatively restricting it:

Definition 19 (Case Distinction). *If $g_\varepsilon : \mathbb{W}^k \to \mathbb{P}_\mathbb{W}$ and for every $a \in \Sigma$, $g_a : \mathbb{W}^{k+1} \to \mathbb{P}_\mathbb{W}$, the function $h : \mathbb{W}^{k+1} \to \mathbb{P}_\mathbb{W}$ such that $h(\varepsilon, \mathbf{y}) = g_\varepsilon(\mathbf{y})$ and $h(a \cdot w, \mathbf{y}) = g_a(w, \mathbf{y})$ is said to be defined by* case distinction *from g_ε and $\{g_a\}_{a\in\Sigma}$ and is denoted as $case(g_\varepsilon, \{g_a\}_{a\in\Sigma})$.*

The idea behind tiering consists in working with denumerable many copies of the underlying algebra $\mathbb{W}$, each indexed by a natural number $n \in \mathbb{N}$ and denoted by $\mathbb{W}_n$. Judgments take the form $f \triangleright \mathbb{W}_{n_1} \times \ldots \times \mathbb{W}_{n_k} \to \mathbb{W}_m$, where $f : \mathbb{W}^k \to \mathbb{W}$. In the following, with slight abuse of notation, $\mathbf{W}$ stands for any expression in the form $\mathbb{W}_{i_1} \times \cdots \times \mathbb{W}_{i_j}$.

Typing rules are in Figure 3. The idea here is that, when generating functions

$$\frac{}{\varepsilon \triangleright \mathbb{W}_k \to \mathbb{W}_k} \qquad \frac{}{r_a \triangleright \mathbb{W}_k \to \mathbb{W}_k} \qquad \frac{}{c_a \triangleright \mathbb{W}_k \to \mathbb{W}_k} \qquad \frac{}{\Pi^k_m \triangleright \mathbb{W}_{n_1} \times \cdots \times \mathbb{W}_{n_k} \to \mathbb{W}_{n_m}}$$

$$\frac{\{g_i \triangleright \mathbb{W}_{s_1} \times \cdots \times \mathbb{W}_{s_r} \to \mathbb{W}_{m_i}\}_{1\leq i\leq p} \qquad f \triangleright \mathbb{W}_{m_1} \times \cdots \times \mathbb{W}_{m_p} \to \mathbb{W}_l}{f \odot (g_1, \ldots, g_l) \triangleright \mathbb{W}_{s_1} \times \cdots \times \mathbb{W}_{s_r} \to \mathbb{W}_l}$$

$$\frac{g_\varepsilon \triangleright \mathbf{W} \to \mathbb{W}_l \qquad \{g_a \triangleright \mathbb{W}_k \times \mathbf{W} \to \mathbb{W}_l\}_{a\in\Sigma}}{case(g_\varepsilon, \{g_a\}_{a\in\Sigma}) \triangleright \mathbb{W}_k \times \mathbf{W} \to \mathbb{W}_l} \qquad \frac{g_\varepsilon \triangleright \mathbf{W} \to \mathbb{W}_k \qquad m > k \qquad \{g_a \triangleright \mathbb{W}_k \times \mathbb{W}_m \times \mathbf{W} \to \mathbb{W}_k\}_{a\in\Sigma}}{rec(g_\epsilon, \{g_a\}_{a\in\Sigma}) \triangleright \mathbb{W}_m \times \mathbf{W} \to \mathbb{W}_k}$$

Fig. 3. Tiering as a Typing System

by primitive recursion, one goes from a level (tier) m for the domain to a *strictly* lower level k for the result. This predicative constraint ensures that recursion does not cause any exponential blowup, simply because the way one can *nest* primitive recursive definitions one inside the other is severely restricted. Please notice that case distinction, although being typed in a similar way, does *not* require the same constraints.

Those probabilistic functions $f : \mathbb{W}^k \to \mathbb{P}_\mathbb{W}$ such that f can be given a type through the rules in Figure 3 are said to be *predicatively recursive*. The class of all predicatively recursive functions is $\mathscr{PT}$. Actually, the class coincides with the one of probabilistic functions which can be computed by PTMs in polynomial time:

Theorem 2. $\mathscr{PT} = \mathscr{PPC}$.

We don't have enough space to give the details of the proof of Theorem 2. It however proceeds essentially by showing the following four lemmas, from which the thesis can be easily inferred:

- On the one hand, one can prove, by a careful encoding, that a form of *simultaneous primitive recursion* is available in predicative recursion.
- On the other hand, PTMs can be shown equivalent, in terms of expressivity, to probabilistic *register* machines; going through register machines has the advantage of facilitating the last two steps.
- Thirdly, any function definable by predicative recurrence can be proved computable by a polytime probabilistic register machine.
- Lastly, one can give an embedding of any polytime probabilistic register machine into a predicatively recursive function, making use of simultaneous recurrence.

Characterizing complexity classes of *probabilistic* functions allows to deal implicitly with concepts like that of a *polynomial time samplable* distribution [2, 8], which is a family $\{\mathcal{D}_n\}_{n\in\mathbb{N}}$ of distributions on strings such that a polytime randomized algorithm produces $\mathcal{D}_n$ when fed with the string 1^n. By Theorem 2, each of them is computed by a function in $\mathscr{PT}$ and, conversely, any predicatively recursive probabilistic function computes one such family.

4 Conclusions

In this paper we make a first step in the direction of characterizing probabilistic computation in itself, from a recursion-theoretical perspective, without reducing it to deterministic computation. The significance of this study is genuinely foundational: working with probabilistic functions allows us to better understand the nature of probabilistic computation on the one hand, but also to study the implicit complexity of a generalization of Leivant's predicative recurrence, all in a unified framework.

More specifically, we give a characterization of computable probabilistic functions by a natural generalization of Kleene's partial recursive functions which includes, among initial functions, one that returns the uniform distribution on $\{0, 1\}$. We then prove the equi-expressivity of the obtained algebra and the class of functions computed by PTMs. In the the second part of the paper, we investigate the relations existing between our recursion-theoretical framework and sub-recursive classes, in the spirit of ICC. More precisely, endowing predicative recurrence with a random base function is proved to lead to a characterization of polynomial-time computable probabilistic functions.

An interesting direction for future work could be the extension of our recursion-theoretic framework to *quantum* computation. In this case one should consider transformations on Hilbert spaces as the basic elements of the computation domain. The main difficulty towards obtaining a completeness result for the resulting algebra and proving the equivalence with quantum Turing machines seems to be the definition of suitable recursion and minimization operators generalizing

the ones described in this paper, given that qubits (the quantum analogues of classical bits) cannot be copied nor erased.

References

1. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational complexity*, 2(2):97–110, 1992.
2. A. Bogdanov and L. Trevisan. Average-case complexity. *Foundations and Trends in Theoretical Computer Science*, 2(1), 2006.
3. U. Dal Lago and P. P. Toldin. A higher-order characterization of probabilistic polynomial time. In *FOPARA*, volume 7177 of *LNCS*, pages 1–18. Springer, 2012.
4. U. Dal Lago and S. Zuppiroli. Probabilistic recursion theory and implicit computational complexity (long version). `http://arxiv.org/abs/1406.3378`, 2014.
5. K. De Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro. Computability by probabilistic machines. *Automata studies*, 34:183–198, 1956.
6. J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.
7. J.-Y. Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.
8. O. Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2000.
9. S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.
10. J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.
11. S. C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742, 1936.
12. D. Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Springer, 1995.
13. D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. *Fundam. Inform.*, 19(1/2):167–184, 1993.
14. M. O. Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963.
15. M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, 1959.
16. E. S. Santos. Probabilistic Turing machines and computability. *Proceedings of the American Mathematical Society*, 22(3):704–710, 1969.
17. E. S. Santos. Computability by probabilistic turing machines. *Transactions of the American Mathematical Society*, 159:165–184, 1971.
18. R. I. Soare. *Recursively enumerable sets and degrees: a study of computable functions and computably generated sets*. Perspectives in mathematical logic. Springer-Verlag, 1987.